Ausbildung Fachinformatiker Anwendungsentwicklung

Programmierübung (mit Musterlösung)

C–Programmierung / Datenstrukturen

1 Aufgabe

Es ist ein Programm (`shakespeare.c`) zu schreiben, das den Namen einer Text–Datei/Pfad

1. als Parameter einliest,
2. diese Datei öffnet,
3. deren Inhalt einliest,
4. bestimmte Operationen ausführt und
5. das Ergebnis in die Standard–Ausgabe schreibt.

Der Datei/Pfadname wird als Argument für die Option *-f* übergeben. Weitere optionale Parameter müssen von dem Programm eingelesen werden können. Die Parameter bestehen jeweils aus einem Buchstaben und sind durch ein vorangestelltes `-` gekennzeichnet. Diese Parameter bestimmen, wie die u.g. Operationen ausgeführt werden sollen. Beim optionalen Parameter `-h` soll lediglich eine `usage()`–Meldung ausgegeben werden (`./shakespeare [-h] [-s <n>] [-l <m>] [-f <inputfile>]`). Folgende Parameter müssen behandelt werden können:

Parameter	Funktion	Ausgabe
-h	mache garnichts	`usage()` ausgeben
-f <inputfile>	Datei einlesen	keine
-s <n>	Suchstringlänge	default = 1
-l <n>	Ausgabelänge	default = Anzahl eingelesener Zeichen

Das Programm soll so konzipiert werden, dass später gegebenenfalls weitere optionale Parameter leicht hinzuzufügen sind (z.B. der initiale Suchstring mit `-a <Anfang>` oder die Version mit `-v`). Außerdem soll das Programm als *Filter* arbeiten können, d.h. falls kein Datei oder Pfadname als Parameter angegeben wurde, soll das Programm seine Eingabe aus dem Kanal Standard–Eingabe (`stdin`) beziehen. Das Programm soll seine Ausgabe immer in die Standard–Ausgabe (`stdout`) schreiben. Einige mögliche Aufrufe könnten z.B. folgendermaßen aussehen:

```
bash$> ./shakespeare ./7zara10.txt
bash$> ./shakespeare -s 3 -l 1000 ./7zara10.txt
bash$> ./shakespeare -s5 -l500 ./7zara10.txt
bash$> ./shakespeare -s 8 -f ./7zara10.txt
bash$> ./shakespeare  -h
bash$> cat ./7zara10.txt | ./shakespeare -s 8
bash$> ./shakespeare -s4 -l1000 < ./7zara10.txt
```

Zum Programm soll eine etwa 4–seitige Dokumentation erstellt werden, in der die Arbeitsweise und der Aufbau des Programms kurz skizziert wird. Dabei sollte auch auf die verwendeten Datenstrukturen eingegangen werden.

Eine mögliche Ausgabe (der anfängliche Suchstring steht in spitzen Klammern) könnte ungefähr so aussehen:

```
bash$> ./shakespeare -s6 -l400 ./7zara10.txt
   <Friedr>ich Nichts; du so langsam, was auf deinem Auge. Dass du mir und Klapper mit
    Vorsicht. Aber ein Dichternen wie sieben dem als an Hoefen? Ein Arzt? Oder boese
    Blick _aller_ Dinge sie, die sich "die␣Lust.␣Lust␣nannte␣keine␣Gipfel␣zu␣neuen␣
    Frevelhaften␣Vortheil." Ungerechtig auf und hinweg und rauchten sie mich selber
    abwendet. Diess nach Vater, antwortete der Erde an und Grauen wissen auch deinetwill
bash$> ./shakespeare -s6 -l400 ./7zara10.txt
   <Friedr>ich Nichts lebt nicht es mir fragt der wieder und Entsagung. Sonderlich.
    Kein Hunger. Oft kommt er mir, so will es seiner Kuh: welche wieder _hoffende. Und
    wer werde vor Maechtigen: ‘Ja, ich mit _meine_ Sonnen-Pfeile, ein faulichkeit die
    den letzten Stuhl in die Wahrsager und tanzen macht mir dieser Vogels. Aber sein -
    und ich hart an seinem Hausthiere mieden, eine Mittel zu Ende zum
bash$> ./shakespeare -s11 -l400 ./7zara10.txt
   <Friedrich N>ietzsche
   Also sprach Zarathustra von der Stelle hin, woher die Stimme kam, und sahen zu ihm
    hinauf. Solchergestalt waren sie Alle: ihr Wahnsinn in der Liebe.
   Licht bin ich: ach, dass die Macht gnaedig wird und herabkommt in’s␣Sichtbare:␣
    Schoenheit␣waechst,␣was␣Einer␣mit␣seinem␣Blute,␣Zarathustra;␣noch␣nahmst␣du␣meinen␣
    Dank␣nicht␣an!␣Ward␣meine␣Welt␣nicht␣eben␣vollkommener␣als␣einem␣Vater,␣am␣aehn
```

2 Funktionen der Parameter

Das Programm soll einen neuen Zufallstext ausgeben, der aus dem eingelesenen Text folgendermaßen berechnet wird:

-s <n> Mit diesem Parameter wird die Suchstringlänge angegeben. Es wird nach einem Suchstring dieser angegebenen Länge im eingelesenen Text gesucht. Zuvor wird ein String dieser Länge aus dem eingelesenen Text ausgewählt (die ersten Zeichen). Dieser String ist dann der *Suchstring*.

Dann wird dieser String im gesamten Text gesucht. Der Folgebuchstabe hinter diesem String wird sich gemerkt und für jeden gefundenden Folgebuchstabe wird gezählt, wie oft er hinter diesem Suchstring im Text vorkommt. Dann wird hinter den Suchstring ein neuer Folgebuchstabe angehängt, der aus der Menge aller gefundenen Folgebuchstabe zufällig, aber unter Berücksichtigung der Wahrscheinlichkeit des Vorkommens, ausgewählt wird.

Anschließend wird bei dem um ein Zeichen verlängerte Suchstring das erste Zeichen nicht beachtet. Der neue Suchstring mit dem angefügten Zeichen hinten und dem vernachlässigten Zeichen vorne hat dann die gleiche Länge wie der anfängliche Suchstring. Mit diesem neuen Suchstring wird das Verfahren wiederholt.

Damit lässt sich ein neuer Ausgabestring erzeugen, der einen Zufallstext erzeugt, der sich aus dem eingelesenen Text zufällig ergibt.

Je kürzer der Suchstring ist, desto unsinniger wird der ausgegebene Text sein. Bei einer Länge von mehr als 5 Zeichen werden schon oft ganze Wörter berücksichtigt und der vermeintliche Sinn des

ausgegebenen Textes erscheint mehr und mehr vorhanden. Ein Beispiel für eine Suchstringlänge von 5 ($\langle abcde \rangle$) würde folgendermaßen aussehen:

$$\langle abcde \rangle_1 + [x_1] \Rightarrow \langle bcdex_1 \rangle_2 + [x_2] \Rightarrow \langle cdex_1x_2 \rangle_3 + [x_3] \ldots$$

Dabei ist $[x_i]$ jeweils ein aus der Wahrscheinlichkeitsverteilung gewählte Buchstabe, der als Folgebuchstabe zum aktuellen Suchstring $\langle \ldots \rangle_i$ im gesamten Text gefunden wurde.

-l <n> Mit diesem Parameter wird die Anzahl der Zeichen angegeben, die vom Programm ausgegeben werden sollen. Der *default*–Wert ist der gesamte Eingabetext. Falls dieser aber sehr groß ist und man nur einen Text begrenzter Länge benötigt, kann man mit diesem Parameter die Länge angeben. Der Parameter muß aber nicht unbedingt die Ausgabe begrenzen, er kann sie auch erweitern, da der generierte Text ja zufällig gebildet wird (eine Begrenzug des auszugebenden Textes sollte durch die vom Compiler größte maximal darstellbare Integer–Zahl erfolgen).

3 Bemerkungen und Tips

Die Entwicklungsumgebung ist *MinGW*. Getestet wird das Programm am günstigsten in einer *Git–bash*.

Das Einlesen der Parameter, Datei und die Verwendung als Filter sollte keine Probleme darstellen.

Eine Herausforderung stellt die Erarbeitung des Algorithmus mit den dazugehörigen Datenstrukturen dar. Daher ist es vorteilhaft, erst das Gerüst zur Behandlung der Parameter und das Einlesen der Textdatei auszuprogrammieren und zu testen.

Anschließend versuchen Sie einen Algorithmus und eine Datenstruktur zu finden, die die Anforderungen erfüllen. Bei allen Überlegungen sollte auch immer der benötigte Speicherplatz und die Laufzeit des Programmes im Auge behalten werden. Nutzen Sie Ihr Wissen über Datenstrukturen, um einen effektiven Algorithmus zu finden. Hier muss nicht nur programmiert werden, sondern es muss auch ein Konzept, die Datenstruktur und ein Algorithmus geplant werden.

Testen Sie mit einer überschaubaren kleinen Eingabedatei, bei der sich die Ausgabe leicht überprüfen lässt.

Fassen Sie die nur für die Datenstrukturen benötigten Hilfsprogramme und Datenstrukturen in externen Dateien zusammen, um die Übersicht zu behalten (Modularisierung). Da diese Aufgabe ein wenig komplexer als die vorherige ist, empfiehlt es sich hier besonders, einzelne Funktions–Komponenten gut gegeneinander abzugrenzen und separat zu testen. Nutzen Sie selbst erstellte Skizzen und Programmablaufpläne. Ein Programmieren *aus dem Bauch heraus* könnte leicht zu einem Durcheinander führen.

4 Hintergrundinformationen

Das Problem gehört zu einer Klasse von Problemen, bei dem ein Zustand durch einen festen Algorithmus immer wieder in einen neuen Zustand überführt wird. Die Menge der möglichen Zustände ist dabei fest.

Konkret ist hier die Menge der möglichen Zustände die Menge aller Möglichkeiten, einen String in der Länge des Suchstrings zu erzeugen. Bei einer Alphabetgröße z.B. von 50 ist die

$$(\text{Menge aller möglichen Zustände für einen String der Länge } s) = 50^s$$

.

Mathematisch läßt sich so ein Problem durch einen sogenannten *Markov–Prozeß* (*Markov–Kette*) beschreiben. Dabei wird aus einem Zustandsvektor (geordnete Menge aller möglichen Zustände: V_i^{alt}) durch Multiplikation mit einer Übergangsmatrix (enthält die Wahrscheinlichkeiten für die einzelnen Übergänge: P_{ij}) ein neuer Zustandsvektor (V_i^{neu}) erzeugt.) (

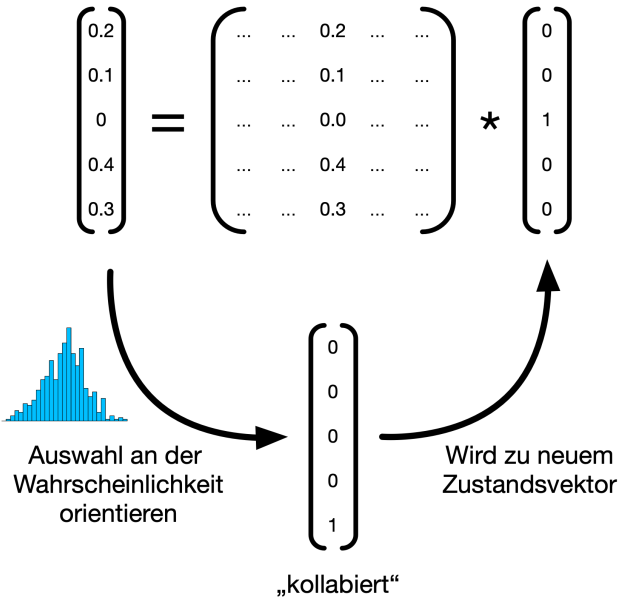
$$V_j^{\text{neu}} = P_{ij} * V_i^{\text{alt}}$$

Für dieses konkrete Problem sähen die Zustandsvektoren für einen 3–stelligen Suchstring etwa so aus:

$$V_i = \begin{pmatrix} P(AAA) \\ P(AAB) \\ P(ABA) \\ \cdots \\ P(DER) \\ \cdots \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \cdots \\ 1 \\ \cdots \end{pmatrix}$$

Aus diesem Zustandsvektor geht dann mit gewissen Wahrscheinlichkeiten ein neuer Zustandsvektor hervor. Die Wahrscheinlichkeiten sind durch die Häufigkeiten der Folgebuchstaben für den jeweiligen String definiert.

Der Index des Zustandsvektors kennzeichnet also den konkreten String. Steht dort bei V^{alt} eine 1, so ist dieser String gerade der aktuelle Suchstring. Aus diesem Zustandsvektor wird ein neuer Zustandsvektor, der bei maximal 50 Indizes (weil nur 50 verschiedene neue Buchstaben angehängt werden können) jeweils eine Zahl zwischen 0 und 1 besitzt. Diese Zahl repräsentiert die Wahrscheinlichkeit für die Wahl dieses Strings als den neuen Suchstring (alle Zahlen müssen sich daher zu 1 addieren). Mit diesen Wahrscheinlichkeiten wird ein neuer Suchstring ausgewählt (dort erscheint jetzt eine 1 und alle anderen Zeilen im Zustandsvektor werden zu 0) und der Prozess wiederholt sich. Dabei bleibt die Übergangsmatrix dieselbe, da sie nur von dem Originaltext abhängt.
Damit wäre der Algorithmus zumindest theoretisch vorgegeben.



Man müsste nur die Matrix für die Übergangswahrscheinlichkeiten aus dem Text errechnen und dann für jeden neuen Zufallsbuchstaben die Matrix mit dem Zustandsvektor multiplizieren. Dann erhält man einen neuen String mit den zugeordneten Wahrscheinlichkeiten und wählt einen gemäß der Wahrscheinlichkeit aus.

Praktisch ist das aber so nicht möglich da:

- Bei einer etwas größeren Länge für den Suchstring (wo es interessant wird: $s > 5$) wird der Zufallsvektor sehr groß. Für $s = 5$ und einem 50–Zeichen-Alphabet ergibt sich eine Zustandsanzahl von 312500000. Eine Suchstringlänge von 10 ergibt z.B. schon eine Zustandsanzahl von 97656250000000000.
- Der Zustandsvektor enthält bei einem 50–Zeichen–Alphabet nur maximal 50 Einträge ungleich Null. Damit besteht er fast nur aus Nullen.

- Die Übergangsmatrix, die die Wahrscheinlichkeiten enthält ist von der Größe des Quadrats der Größe der Zustandsvektoren und entsprechend groß.
- Abgesehen von dem astronomisch hohen Speicherbedarf würde die Berechnung bei diesen Größen entsprechend lange dauern.
- Wird der Suchstring um ein Zeichen erhöht, so würde der Speicherbedarf für den Zustandsvektor um den Faktor 50 wachsen und der Speicherbedarf für die Übergangsmatrix um den Faktor $50*50 = 2500$. Man sagt hier, dass der Algorithmus schlecht mit der Eingangsgröße *skaliert* (exponentiell). Dies wird mit der sogenannten *O–Notation* ausgedrückt, die angibt in welcher Stärke der Speicherbedarf bzw. die Laufzeit mit der Eingangsgröße wächst. Hier hätten wir ein exponentielles Ansteigen: $O(k^n), k > 2$). Für Algorithmen strebt man (abgesehen von $O(1)$) eine Komplexität von $O(log(n))$ oder $O(n)$, schlimmstenfalls $O(n^2)$ an.

Obwohl wir hier eigentlich einen klaren Algorithmus hätten , der einfach ausprogrammiert werden könnte (und auf 1– und 2–dimensionalen Feldern/Arrays beruht), müssen wir nach einem praktischeren Algorithmus suchen, der mit dem zur Verfügung stehenden Speicher auskommt und ihn effizienter nutzt!

5 Abgabe

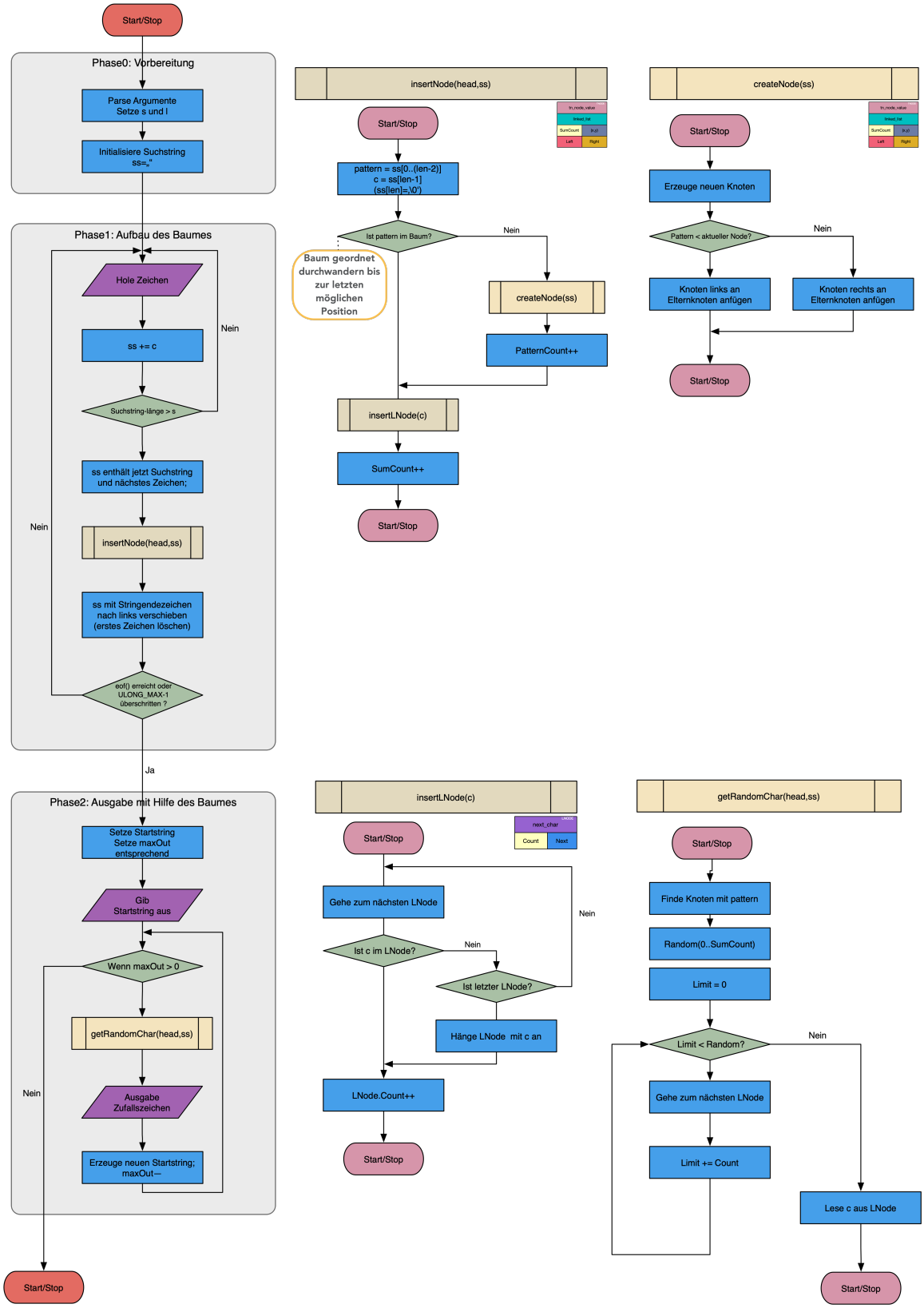
Die lauffähigen Programme und der kommentierte Quelltext sind mit einer etwa 4–seitigen Dokumentation abzugeben (Email, Dokumentation in HTML,TEX oder `ASCII`).

6 Zeitrahmen

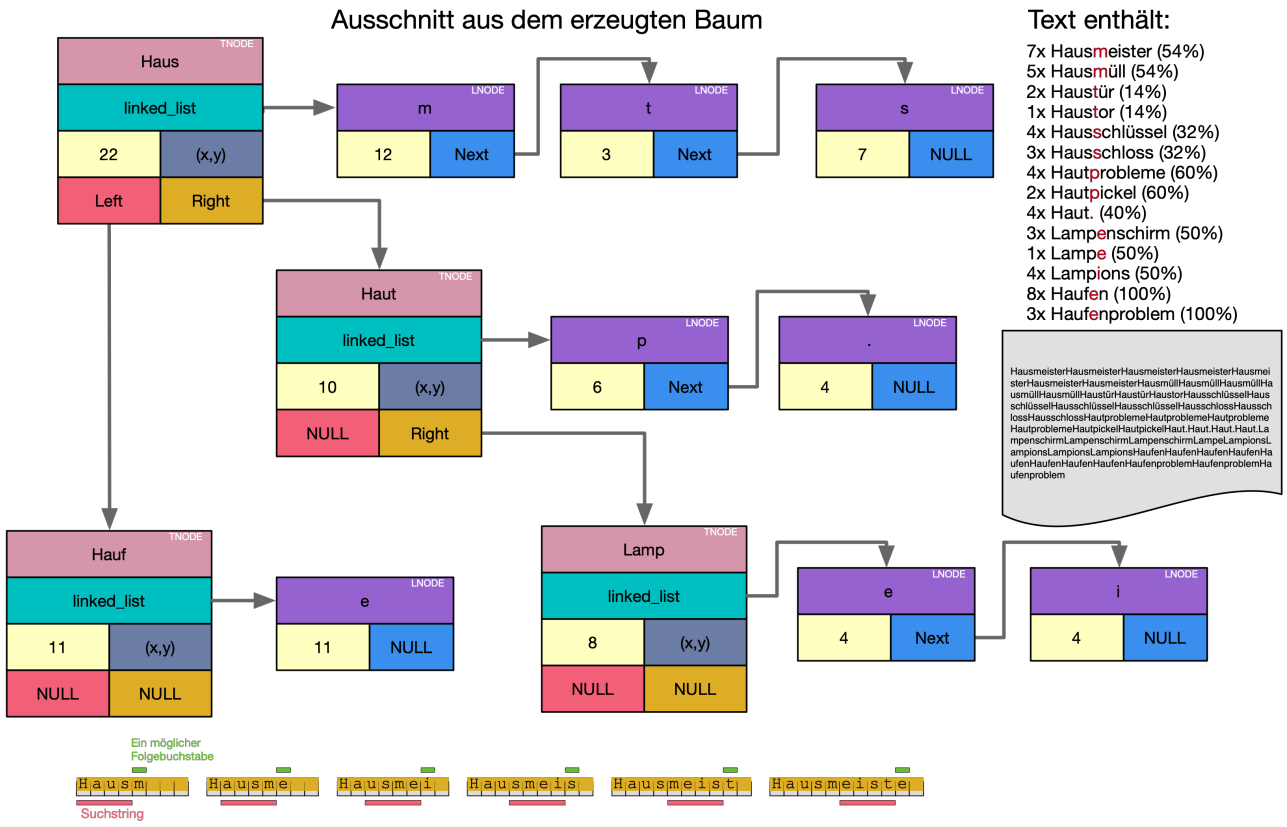
Der Zeitrahmen dieser Aufgabe ist mit etwa 60 Stunden angesetzt.

7 Musterlösung

Der Programmablaufplan:



Die Datenstruktur zum Abspeichern der eingelesenen Information und der Generierung des Zufallstextes:



Die Auswahl des zufällig bestimmten Buchstabens aus den gewichteten Wahrscheinlichkeiten kann folgendermaßen erfolgen:

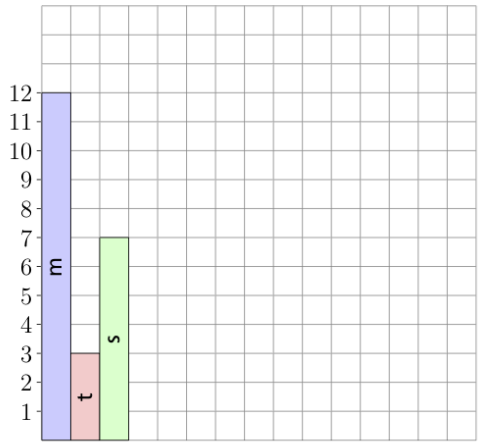
Fachinformatiker Anwendungsentwickler (1. Ausbildungsjahr 2020) *Tipp: Aufgabe2*

Tipp für Aufgabe 2

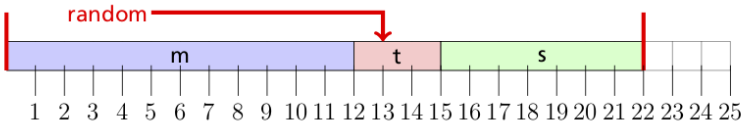
In der Aufgabe 2 soll ein Buchstabe aus einer Verteilung von verschiedenen Buchstaben gewählt werden. Dabei soll die Häufigkeit der vorkommenden Buchstaben berücksichtigt werden, aber ansonsten der ausgewählte Buchstabe zufällig gewählt werden.

Eine Lösung für diese Aufgabe, einerseits die Häufigkeiten zu berücksichtigen und andererseits aber einen Buchstaben zufällig auszuwählen, sei hier kurz skizziert.

Nehmen wir an wir hätten 3 verschiedene Buchstaben b_1, b_2 und b_3 mit den Häufikeiten h_1, h_2 und h_3. In der Datenstruktur–Grafik wären das z.b. m, t und s mit den Häufigkeiten 12, 3 und 7.



Wenn man die Häufigkeiten als Intervalle betrachtet und nebeneinander anordnet, bekommt man ein 3–geteiltes Intervall von der Länge 22.

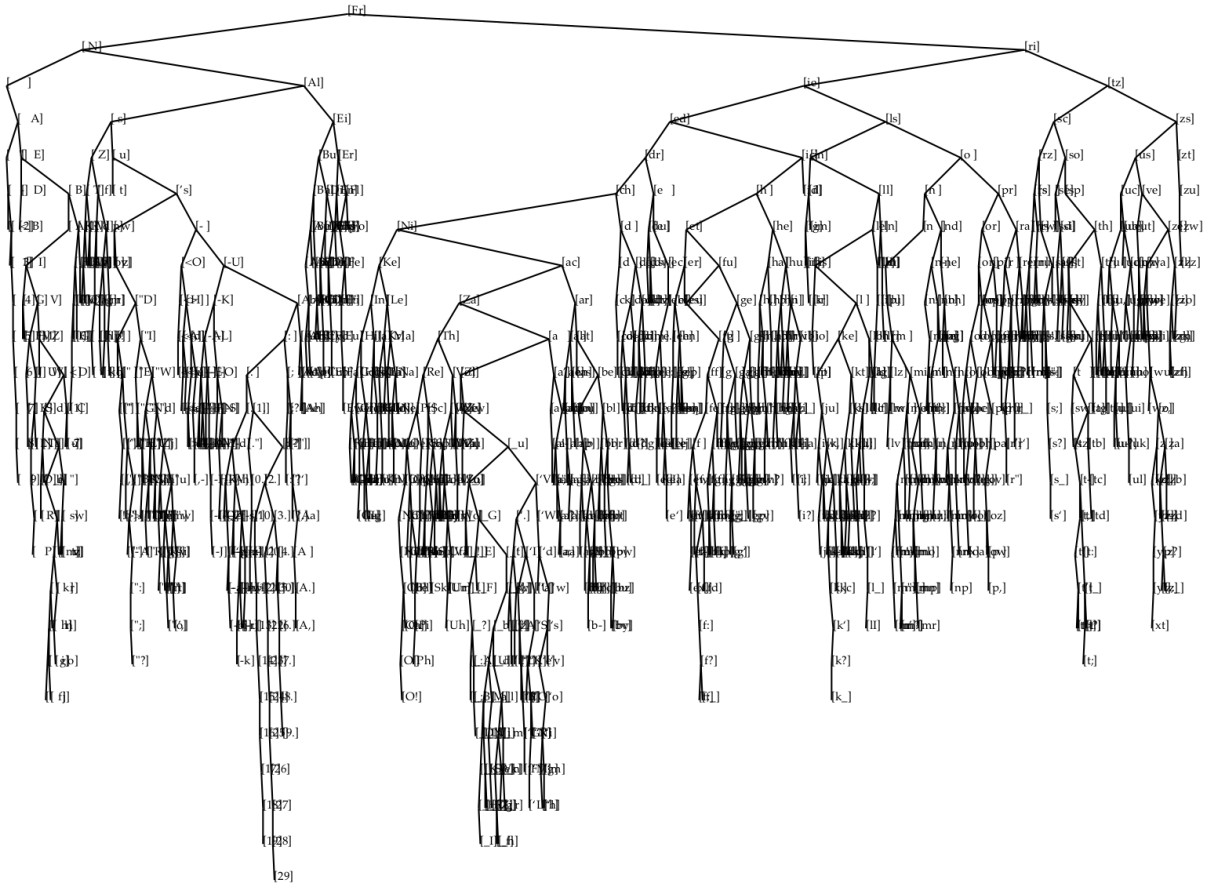


Bestimmt man nun eine Zufallszahl im Bereich $0 \ldots 22$ und berechnet[1], in welches Intervall sie fällt, so hat man die Größe der Intervalle bei der Auswahl der Zufallszahl berücksichtigt.

[1] in Pseudo-Code z.B.

```
r=random(0..22);
i=0;
while(Intervallende[i]<r)(i++;);
wähle Buchstabe aus Intervall[i];
```

Die Musterlösung enthält auch einen optionalen Parameter `-d` für das Programm, der nicht gefordert wurde aber zur leichteren Übersicht mit in die Musterlösung eingebaut wurde. Der Schalter `-d7` druckt den erstellten Baum (hier mit `-s2`) u.a. auch als Baumstruktur im PostScript–Format.



Musterlösung für `shakespeare.c`

```c
/**
 * shakespeare.c
 * Author: Frank Zimmermann
 * 9/2020
 *
 * Generates a random text from an input text
 * uses tree and linked list as data structures
 *
 */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h> /*C-99*/
#include <time.h>
#include <limits.h>

#ifndef BINTREE
#include "bintree.h"
#endif

#define ERROR_FILEOPEN(message) fprintf(stderr, "Konnte␣Datei␣%s␣nicht␣öffnen!\n", (message));
#define ERROR_PARAMETER() fprintf(stderr, "Wrong␣command␣line␣...\n");
#define ERROR_SSLENGTH(message1, message2) fprintf(stderr,\
       "Die␣Suchstringlänge␣von␣%d␣überschreitet␣die␣maximale␣Länge␣und␣wurde␣auf␣%d␣gesetzt\n",\
       (message1), (message2));
#define ERROR_USAGE(message) fprintf(stderr, \
       "Usage:␣%s␣[-h|-s<PatternLength:1-50>|-l<OutputLength>|-f<FileName>|-d<Debug:0-7>]\n",\
       (message));
#define DEBUG_PATTERNCOUNT(message) printf("There␣were␣%d␣different␣patterns\n", (message));

/* some global variables ************************************/
char ss[MAXPAT];                /* buffer for text ss=searchstring+c+'\0' */
int ssLength = 2;               /* default ss length =searchstring+c */
TNODE *head;                    /* pointer to root of binary tree */
char search[MAXPAT] = "";       /* the current ss that determines the next char */
int patternCount = 0;           /* DEBUG: count the different patterns (no of TNODEs) */

int main(int argc, char *argv[]) {
  int debug = 0;                /* default: no debugging */
  char opt;                     /* holds the current option char */
  char c;                       /* char read in */
  int countFirstChar;           /* needed for fill the first pattern */
  int mainReturn = EXIT_SUCCESS; /* indicates errors in command line */
  bool isEnough;                /* flag for completeness of first pattern */
  int i;
  unsigned long maxOut = 0;     /* max number of output characters */
  unsigned long totalCharCount = 0; /* counter for input chars */

  srand((unsigned)time(NULL));  /* initialize random function */

  head = NULL;                  /* initialiaze root tree pointer */

  /* Parsing the commandline */
  while ((opt = getopt(argc, argv, "s:f:l:d:h")) != -1) {
    switch (opt) {
    case 'h': /* Usage */
```

```
59      ERROR_USAGE(argv[0]);
60      exit(EXIT_FAILURE);
61      break;
62    case 's':                     /* length of searchstring */
63      ssLength = (int)atoi(optarg) + 1; /* ss = searchstring + c */
64      break;
65    case 'l':                     /* number of max char to generate */
66      maxOut = (int)atoi(optarg); /* besser: überprüfen! */
67      break;
68    case 'd':                     /* 0=no Debug, 1=print ss, 2=print tree nodes + list values */
69      debug = (int)atoi(optarg); /* 4=print some levels of graphic tree in Postscript */
70      if (debug > 7 || debug < 0) {
71        debug = 0;
72      }
73      break;
74    case 'f':
75      if (!freopen(optarg, "r", stdin)) {
76        /* to use as a filter as well as with filename */
77        ERROR_FILEOPEN(optarg);
78        ERROR_USAGE(argv[0]);
79        exit(EXIT_FAILURE);
80      }
81      break;
82    }
83  }
84
85  /** Here we _would_ test for further arguments... */
86  /** ...but we are lazy */
87
88  if (mainReturn == EXIT_FAILURE) {
89    /* program call was faulty */
90    /* for future checks */
91    ERROR_PARAMETER();
92    exit(EXIT_FAILURE); /* exit program with error code */
93  }
94
95  if (ssLength > MAXPAT - 1) {
96    /* to avoid buffer overflow */
97    ERROR_SSLENGTH(ssLength - 1, MAXPAT - 2)
98    ssLength = MAXPAT - 1;
99  }
100
101 /* Reading in the text char by char */
102 /* ********************************************************************** */
103
104 countFirstChar = 0;
105 isEnough = false;
106 ss[ssLength] = '\0'; /* string termination */
107 while ((!feof(stdin)) && ((c = fgetc(stdin)) != EOF)) {
108   if (isEnough == true) {/* the current char completes ss (searchstr + c) */
109     ss[ssLength - 1] = c;
110     head = insert(head, ss); /* save the ss in the tree */
111     if (debug & 1) {
112       printf("%s\n", ss);
113     }; /* print ss */
114     for (i = 0; i < ssLength; i++) { /* shift to left ... */
115       ss[i] = ss[i + 1];
116     }
117   } /* ...and get the next char...*/
118   else { /* filling up the ss...until searchString is full */
119     ss[countFirstChar++] = c;
```

```
120       isEnough = (countFirstChar == ssLength - 1) ? true : false;
121     }
122     if (totalCharCount++ == (ULONG_MAX - 1)) {
123       /* watch if chars are longer than ULONG_MAX */
124       break;
125     }
126   }
127   fclose(stdin);
128
129   /* DEBUG */
130   if (debug & 2) {
131     DEBUG_PATTERNCOUNT(patternCount);
132     tree_print(head); /* print the tree with linked list */
133   }
134
135   /* print out the generated text char by char */
136   /* when not otherwise set start at beginning */
137   if (strcmp(search, "") == 0) {
138     strcpy(search, head->sPattern);
139   }
140
141   /* DEBUG */
142   if (debug & 4) {
143     print_ps(head);
144   }
145
146   /* if maxOut not set use the input text length */
147   maxOut = (maxOut == 0) ? totalCharCount : maxOut;
148
149   printf("<%s>", search); /* ...beginning with starting pattern */
150   /* maxOut = (maxOut >= strlen(search)) ? (maxOut -= strlen(search)) : 0; unsequenced ! */
151   if(maxOut >= strlen(search)){
152     maxOut -= strlen(search);
153   }
154   else{
155     maxOut = 0;
156   }
157   while (maxOut-- > 0) {
158     printf("%c", random_text(head)); /* print out next generated char */
159   }
160
161   printf("\n");
162   return mainReturn;
163 }
```

Musterlösung für `bintree.c`

```
1  /**
2   * Utility code for shakespeare.c: for tree, linked list and debugging
3   * Author: Frank Zimmermann
4   * Date: 15.10.2020
5   * modified: 29.10.24
6   */
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <string.h>
10
11 #ifndef BINTREE
```

```
12 #include "bintree.h"
13 #endif
14
15 #define ERROR_MEMORYALLOCATION fprintf(stderr, "Konnte␣keinen␣Speicher␣allokieren!\n")
16
17 extern char ss;        /* buffer for text ss */
18 extern int ssLength;   /* default pattern length */
19 extern TNODE *head;    /* pointer to root of binary tree */
20 extern char search[]; /* the current ss that determines the next char */
21 extern int patternCount; /* count the different patterns */
22
23 int psHeight = 0;
24 int psWidth = 0;
25 int psXconst = 15;
26 int psYconst = 15;
27 int maxY = 0;
28
29 POINT position = {0, 0}; /* position in tree */
30
31 /* Inserts the ss into the data structure if not present, otherwise increment counter */
32 TNODE *insert(TNODE *pn, char *ss) {
33   int cond;
34   LNODE *pl = NULL;
35   char sPattern[MAXPAT + 1]; /* sPattern will hold search pattern */
36   char c = ss[ssLength - 1]; /* get char before string end */
37   strcpy(sPattern, ss);
38   sPattern[ssLength - 1] = '\0'; /* cut sPattern from string */
39
40   if (pn == NULL) {
41     /* new entry */
42     pn = malloc(sizeof(TNODE));       /* allocating space for new tnode */
43     if(pn==NULL){
44       ERROR_MEMORYALLOCATION;
45       exit(EXIT_FAILURE);
46     }
47     strcpy(pn->sPattern, sPattern);   /* copy search pattern into tnode */
48     pn->pl = insert_lnode(pl, c);     /* put lnode into tnode */
49     pn->pleft = pn->pright = NULL;    /* leave tnode */
50     pn->maxList = 1;                  /* first entry in list */
51     patternCount++;                   /* number of different pattern */
52   } else if ((cond = strcmp(pn->sPattern, sPattern)) == 0) {
53     /* ss is already in tree */
54     insert_lnode(pn->pl, c); /* insert into list or increment counter */
55     pn->maxList++;         /* increment entry counter */
56   } else if (cond > 0) {
57     /* add to the left */
58     pn->pleft = insert(pn->pleft, ss); /* recursive */
59   } else {
60     /* add to the right */
61     pn->pright = insert(pn->pright, ss); /* recursive */
62   }
63   return pn;
64 }
65
66 /* appends a lnode if non existing, otherwise increment counter */
67 LNODE *insert_lnode(LNODE *pl, char c) {
68   if (pl == NULL) {
69     /* new entry */
70     pl = malloc(sizeof(LNODE)); /* allocating space for new lnode */
71     if(pl==NULL){
72       ERROR_MEMORYALLOCATION;
```

```
73      exit(EXIT_FAILURE);
74    }
75    pl->pnext = NULL;                /* end of list */
76    pl->nextChar = c;                /* copy char into nextChar */
77    pl->counter = 1;                 /* init counter */
78  } else if (pl->nextChar == c) {
79    /* c is already in structure */
80    pl->counter++; /* increment counter */
81  } else {
82    pl->pnext = insert_lnode(pl->pnext, c); /* try next, recursive */
83  }
84  return pl;
85 }
86
87 char random_text(TNODE *pn) {
88  /* generates the next char from s */
89  TNODE *pnode;
90  int index;
91  char c = 'z';
92  int i;
93
94  pnode = lookup(pn);           /* searching the tnode that contains s */
95  index = get_index(pnode);     /* random index between 1...maxList */
96  c = get_next_char(pnode, index); /* the generated char */
97  for (i = 0; i < ssLength - 2; i++) {
98    /* correcting the new search pattern */
99    search[i] = search[i + 1];
100 }
101 search[ssLength - 2] = c;
102 search[ssLength - 1] = '\0';
103 return c;
104 }
105
106 char get_next_char(TNODE *pn, int index) {
107  LNODE *pl;
108
109  pl = pn->pl;
110  while ((index -= pl->counter) > 0) {
111    if (pl->pnext != NULL) {
112      pl = pl->pnext;
113    } else {
114      break;
115    }
116  }
117  return pl->nextChar;
118 }
119
120 TNODE *lookup(TNODE *pn) {
121  /* lookup an node entry in head */
122  int cond;
123  TNODE *retval;
124
125  if (pn == NULL) {
126    retval = NULL;
127  }
128  if ((cond = strcmp(pn->sPattern, search)) == 0) {
129    retval = pn;
130  } else if (cond > 0) {
131    retval = lookup(pn->pleft);
132  } else {
133    retval = lookup(pn->pright);
```

```
134   }
135   return retval;
136 }
137
138 void tree_print(TNODE *pn) {
139   /* traverse tree from left to right */
140   if (pn != NULL) {
141     tree_print(pn->pleft);
142     printf("%s␣[%d]->␣", pn->sPattern, pn->maxList);
143     list_print(pn);
144     tree_print(pn->pright);
145   }
146 }
147
148 void print_ps(TNODE *p) {
149   FILE *fout;
150   if ((fout = fopen("shakespeare.ps", "w")) != NULL) {
151     print_header_ps(fout);
152     fill_tree_ps(fout, p, NULL);
153     print_tree_ps(fout, p, NULL);
154     print_footer_ps(fout);
155     fclose(fout);
156   }
157 }
158
159 void list_print(TNODE *pn) {
160   LNODE *pl;
161
162   for (pl = pn->pl; pl != NULL; pl = pl->pnext) {
163     /* move through the linked list...*/
164     printf("(%c,%02d)", pl->nextChar, pl->counter);
165   }
166   printf("\n");
167 }
168
169 int get_index(TNODE *pn) {
170   int rand_index;
171   /* Intervall bestimmen */
172   int a=1;
173   int e=pn->maxList;
174   double range = e-a+1.0;
175   rand_index = a + (int)(range * rand() / (RAND_MAX + 1.0));
176   /* these corrections should never happen */
177   if (rand_index < 1) {
178     rand_index = 1;
179   }
180   if (rand_index > pn->maxList) {
181     rand_index = pn->maxList;
182   }
183   return rand_index;
184 }
185
186 POINT ps_coord(int x, int y) {
187   POINT pos;
188   /* pos.x = x*(842-2*psXconst) / patternCount +psXconst; */
189   pos.x = x * (750 - 2 * psXconst) / patternCount + psXconst;
190   /*  pos.y = (590-psYconst) - (y*(590-psYconst) / maxY) +psYconst; */
191   pos.y = (590 - psYconst) - (y * (590 - psYconst) / maxY) + psYconst;
192   return pos;
193 }
194
```

```
195 void print_header_ps(FILE *fp) {
196   fprintf(fp, "%%!PS\r\n");
197   fprintf(fp, "/inch␣{␣72␣mul␣}␣def\r\n");
198   fprintf(fp, "590␣0␣␣translate\r\n");
199   fprintf(fp, "90␣rotate\r\n");
200 }
201 void print_footer_ps(FILE *fp) {
202   fprintf(fp, "\r\n");
203 }
204
205 void fill_node_ps(FILE *fout, TNODE *pn, TNODE *ppn) {
206   pn->pos.x = position.x++;
207   pn->pos.y = position.y;
208 }
209
210 void print_node_ps(FILE *fout, TNODE *pn, TNODE *ppn) {
211   char str[MAXPAT + 1];
212   char *ptr;
213   if (ppn != NULL) {
214     fprintf(fout, "%d␣%d␣moveto\r\n", ps_coord(ppn->pos.x, ppn->pos.y).x, ps_coord(ppn->pos.x, ppn->pos.y).y);
215     fprintf(fout, "%d␣%d␣lineto\r\n", ps_coord(pn->pos.x, pn->pos.y).x, ps_coord(pn->pos.x, pn->pos.y).y);
216     fprintf(fout, "stroke\r\n");
217   }
218   fprintf(fout, "%d␣%d␣moveto\r\n", ps_coord(pn->pos.x - psWidth, pn->pos.y - psHeight).x,
219           ps_coord(pn->pos.x - psWidth, pn->pos.y - psHeight).y);
220   fprintf(fout, "gsave\r\n");
221   fprintf(fout, "/Palatino-Roman\r\n");
222   fprintf(fout, "0.1␣inch\r\n");
223   fprintf(fout, "selectfont\r\n");
224   strcpy(str, pn->sPattern);
225   while ((ptr = strchr(str, '(')) != NULL) {
226     /* ( and ) have to be escaped for PS */
227     *ptr = '<';
228   }
229   while ((ptr = strchr(str, ')')) != NULL) {
230     *ptr = '>';
231   }
232
233   fprintf(fout, "([%s])␣show\r\n", str);
234   fprintf(fout, "grestore\r\n");
235 }
236
237 void fill_tree_ps(FILE *fout, TNODE *pn, TNODE *ppn) {
238   position.y++;
239   if (pn != NULL) {
240     fill_tree_ps(fout, pn->pleft, pn);
241     fill_node_ps(fout, pn, ppn);
242     fill_tree_ps(fout, pn->pright, pn);
243   }
244   if (maxY < position.y) {
245     maxY = position.y;
246   }
247   position.y--;
248 }
249
250 void print_tree_ps(FILE *fout, TNODE *pn, TNODE *ppn) {
251   position.y++;
252   if (pn != NULL) {
253     print_tree_ps(fout, pn->pleft, pn);
254     print_node_ps(fout, pn, ppn);
255     print_tree_ps(fout, pn->pright, pn);
```

```
256   }
257   position.y--;
258 }
```

Musterlösung für `bintree.h`

```
1  #ifndef BINTREE
2  #define BINTREE
3
4  /* This is the number for length of 'search string' + 'next char' + '\0' */
5  #define MAXPAT 52
6
7  /* A structure used for PS output */
8  struct point {
9    int x;
10   int y;
11 };
12 typedef struct point POINT;
13
14 /* List that contains for a given sPattern the nextChar and its frequency */
15 struct lnode {
16   struct lnode *pnext;     /* points to next lnode entry */
17   char nextChar;           /* contains next char for pattern*/
18   int counter;             /* counts occurences of char for defined pattern*/
19 };
20 typedef struct lnode LNODE;
21
22 /* Node in a binary tree that contains a pattern and a pointer to a List (see above) */
23 struct tnode {
24   struct tnode *pleft;
25   struct tnode *pright;
26   char sPattern[MAXPAT - 1];
27   LNODE* pl;
28   int maxList;             /* counts the entries in the list */
29   POINT pos;
30 };
31 typedef struct tnode TNODE;
32
33 /**
34  * function prototypes
35  */
36 TNODE* insert(TNODE*, char*);
37 LNODE* insert_lnode(LNODE*, char);
38 char   random_text(TNODE*);
39 char   get_next_char(TNODE*, int);
40 TNODE* lookup(TNODE*);
41 void   tree_print(TNODE*);
42 void   print_ps(TNODE*);
43
44 void   list_print(TNODE*);
45 int    get_index(TNODE*);
46 POINT  ps_coord(int, int);
47 void   print_header_ps(FILE*);
48 void   print_footer_ps(FILE*);
49
50 void   fill_node_ps(FILE*, TNODE*, TNODE*);
51 void   print_node_ps(FILE*, TNODE*, TNODE*);
52 void   fill_tree_ps(FILE*, TNODE*, TNODE*);
```

```
53 void  print_tree_ps(FILE*, TNODE*, TNODE*);
54
55 #endif
```

Musterlösung für `Makefile`

```
1  CC = gcc
2  OBJ = shakespeare.o bintree.o
3  CFLAGS= -Wall
4  LATEX = latexmk
5  LATEXFLAGS = -pdf -silent -interaction=nonstopmode
6  TEXFILE = aufgabe02
7  LATEXTMP = *.aux *.log *.listing *.fls *.fdb_latexmk *.synctex.gz
8
9
10 all: aufgabe02.pdf shakespeare
11
12 shakespeare: $(OBJ)
13         $(CC) $(CFLAGS) -o shakespeare shakespeare.o bintree.o
14
15 shakespeare.o: shakespeare.c
16         $(CC) $(CFLAGS) -c shakespeare.c
17
18 bintree.o: bintree.c bintree.h
19         $(CC) $(CFLAGS) -c bintree.c
20
21 aufgabe02.pdf: aufgabe02.tex shakespeare.c bintree.c bintree.h shakespeareDatenstruktur.png \
22         Aufgabe02Flowchart.png TippWahrscheinlichkeiten.png shakespeareBaum2.png Makefile
23         $(LATEX) $(LATEXFLAGS) $(TEXFILE).tex && open $(TEXFILE).pdf
24
25 clean:
26         rm -f $(OBJ) $(LATEXTMP)
```